# Database Programming with Perl

**Kirrily Robert**

**Database Programming with Perl**
by Kirrily Robert



## Open Publications License 1.0

# Table of Contents

# List of Tables

# Chapter 1. Introduction

Welcome to Netizen's *Database Programming with Perl* training course. This is a one-day course in which you will learn how to write database-backed websites using Perl and and the powerful DBI module.

## 1.1. Course outline

- About databases
- Text based ("flat file") databases
- Relational databases
- Tables and relationships
- Structured Query Language (SQL)
- MySQL and other database servers
- Features of MySQL
- Getting MySQL
- Setting up MySQL databases
- The MySQL client
- The DBI and DBD modules
- What is DBI?
- DBI syntax
- DBI exercises
- Extended exercises
- References (optional topic)

## 1.2. Assumed knowledge

It is assumed that you know and understand the following topics:

- Unix - logging in, creating and editing files
- Perl - variable types, operators and functions, conditional constructs, subroutines, basic regular expressions
- Basic database theory - tables, records, fields

If you need help with editing files under Unix, a cheat-sheet is available in Appendix A and an editor command summary in Appendix B. The Unix operating system commands you will need are mentioned and explained very briefly throughout the course - please feel free to ask if you need more help. The required Perl knowledge was covered in Netizen’s "Introduction to Perl" training module. Some of the material taught in "Intermediate Perl" is also useful to this module.

## 1.3. Module objectives

- Understand what a database is and use correct terminology to describe types of databases and parts of databases
- Understand and use flat file or textual databases with Perl
- Understand the advantages and limitations of flat file or textual databases and relational databases
- Understand and use Structured Query Language (SQL) to manipulate data in a relational database
- Know about MySQL and other relational databases suitable for small to medium applications
- Use the MySQL command line client to perform SQL queries

- Understand and use Perl's DBI module to interact with databases
- Use the skills and knowledge learnt in this module to create a sample application

## 1.4. Platform and version details

This module is taught using Unix or a Unix-like operating system. Most of what is learnt will work equally well on Windows NT or other operating systems; your instructor will inform you throughout the course of any areas which differ.

All Netizen's Perl training courses use Perl 5, the most recent major release of the Perl language. Perl 5 differs signficantly from previous versions of Perl, so you will need a Perl 5 interpreter to use what you have learnt. However, older Perl programs should work fine under Perl 5.

The database server used during this module is MySQL, available from http://www.mysql.com. We have chosen it because it is free for most purposes, runs on many platforms, is the most common database used by ISPs offering database services to web hosting clients, and has a good feature set for our purposes. However, all the Perl code examples given in this module will work equally well with any of a number of database systems, including PostgreSQL, Oracle, Sybase, and Informix.

## 1.5. The course notes

These course notes contain material which will guide you through the topics listed above, as well as appendices containing other useful information.

The following typographic conventions are used in these notes:

System commands appear in **this typeface**

Literal text which you should type in to the command line or editor appears as `monospaced font`.

Keystrokes which you should type appear like this: **ENTER**. Combinations of keys appear like this: **CTRL-D**

```
Program listings and other literal listings of what ap-
pears on the
screen appear in a monospaced font like this.
```

Parts of commands or other literal text which should be replaced by your own specific values appears *like this*

Notes and tips appear offset from the text like this.

Notes which are marked "Advanced" are for those who are racing ahead or who already have some knowledge of the topic at hand. The information contained in these notes is not essential to your understanding of the topic, but may be of interest to those who want to extend their knowledge.

Notes marked with "Readme" are pointers to more information which can be found in your textbook or in online documentation such as manual pages or websites.

## 1.6. Other materials

In addition to these notes, you should have a copy of the required text book for this course: Programming Perl (2nd ed.) by Schwartz, Wall, and Christiansen (published by O'Reilly and Associates) -- more commonly referred to as "the Camel book". The Camel book will be used throughout the day, and will be a valuable reference to take home and keep next to your computer.

You will also have received a floppy disk containing these notes in HTML form (with working links to external resources etc) and all the example scripts and data used in this

course.

Lastly, you will have been given a nametag with your name and company on the front, and a username and password on the back.

# Chapter 2. About databases

## 2.1. In this chapter...

This chapter talks about databases in general, and the different types of databases which can be used with Perl.

## 2.2. What is a database?

- A database is a collection of related information.
- The data stored in a database is persistent.

## 2.3. Types of databases

There are many different types of databases, including:

- Flat-file text databases
- Associative flat-file databases such as Berkeley DB
- Relational databases
- Object databases
- Network databases
- Hierarchical databases such as LDAP

Relational databases are by far the most useful type commonly available, and this training module focusses largely on them, after looking briefly at flat file text databases.

## 2.4. Database management systems

A database management system (DBMS) is a collection of software which can be used to create, maintain and work with databases. A client/server database system is one in which the database is stored and managed by a database server, and client software is used to request information from the server or to send commands to the server.

## 2.5. Uses of databases

Databases are commonly used to store bodies of data which are too large to be managed on paper or through simple spreadsheets. Most businesses use databases for accounts, inventory, personnel, and other record keeping. Databases are also becoming more widely used by home users for address books, cd collections, recipe archives, etc. There are very few fields in which databases cannot be used.

## 2.6. Chapter summary

- A database is a collection of related information.
- Data stored in a database is persistent
- There are a number of different types of databases, including flat file, relational, and others
- Database management systems are collections of software used to manage databases
- Databases are widely used in many fields

# Chapter 3. Textfiles as databases

## 3.1. In this chapter...

In this chapter we investigate text-based or "flat file" databases and how to use Perl to manipulate them. We also discuss some of the limitations of this database format.

## 3.2. Delimited text files

A delimited text file is one in which each line of text is a record, and the fields are separated by a known character.

The character used to delimit the data varies according to the type of data. Common delimiters include the tab character (`\t` in Perl) or various punctuation characters. The delimiter should always be one which does not appear in the data.

Delimited text files are easily produced by most desktop spreadsheet and database applications (eg Microsoft Excel, Microsoft Access). You can usually choose "File" then "Save As" or "Export", then select the type of file you would like to save as.

Imagine a file which contains peoples' given names, surnames, and ages, delimited by the pipe (`|`) symbol:

```
Fred|Flintstone|40
Wilma|Flintstone|36
Barney|Rubble|38
Betty|Rubble|34
Homer|Simpson|45
Marge|Simpson|39
Bart|Simpson|11
Lisa|Simpson|9
```

The file above is available in your exercises directory as `delimited.txt`.

## 3.2.1. Reading delimited text files

To read from a delimited text file:

```
#!/usr/bin/perl -w

use strict;

open (INPUT, "delimited.txt") or die "Can’t open data file: $!";

while (<INPUT>) {
        chomp;                  # remove newline
        my @fields = split(/\|/, $_);
        print "$fields[1], $fields[0]: $fields[2]\n";
}

close INPUT;
```

This should print out:

```
Flintstone, Fred: 40
Flintstone, Wilma: 36
...
```

And so on.

## 3.2.2. Searching for records

One of the common uses of databases is to search for specific records.

```
#!/usr/bin/perl -w

use strict;

# Find out what record the user wants:
```

```
print "Search for: ";
chomp (my $search_string = <STDIN>);

open (INPUT, "delimited.txt") or die "Can’t open data file: $!";

while (<INPUT>) {
        chomp;                  # remove newline
        my @fields = split(/\|/, $_);

        # test whether the search string matches given or fam-
ily name
        if ($fields[0] =~ /$search_string/
                or $fields[1] =~ /$search_string/) {
                print "$fields[1], $fields[0]: $fields[2]\n";
        }
}

close INPUT;
```

## 3.2.3. Sorting records

Sorting records from a flat text database can be quite difficult. Simply sorting the items line by line is one simplistic approach:

```
#!/usr/bin/perl -w

use strict;

open (INPUT, "delimited.txt") or die "Can’t open data file: $!";

my @records = sort <INPUT>;

foreach (@records) {
        chomp;                  # remove newline
        my @fields = split(/\|/, $_);
```

```
        print "$fields[1], $fields[0]: $fields[2]\n";
}

close INPUT;
```

The above technique can only sort on the first field of the data (in the case of our example, that would be the given name) and may have difficulties when it encounters the delimiter.

To sort by any other field, we would first need to load the data into a list of lists (using references), then use the `sort()` function's optional first argument to specify a subroutine to use for sorting:

```
#!/usr/bin/perl -w

use strict;

open (INPUT, "delimited.txt") or die "Can't open data file: $!";

while (<INPUT>) {
        chomp;
        my @this_record = split(/\|/, $_);

        # build a list-of-
lists containing references to each record
        push (@records, \@this_record);
}

# sort takes an optional argument of what subrou-
tine to use to sort
# the data...

my @sorted = sort given_name_order @records;

foreach $record (@sorted) {
        # we have to print the items via a refer-
ence to the array...
        print "$record->[1], $record->[0]: $record->[2]\n";
```

```
}

# subroutine to implement sorting order
sub given_name_order {
        $a->[0] cmp $b->[0];
}
```

Obviously this can be quite tricky, especially if the programmer is not totally familiar with Perl references. It also requires loading the entire data set into memory, which would be very inefficient for large databases.

### 3.2.4. Writing to delimited text files

The most useful function for writing to delimited text files is `join`, which is the logical equivalent of `split`.

```
#!/usr/bin/perl -w

use strict;

open OUTPUT, ">>delimited.txt" or die "Can't open out-
put file: $!";

my @record = qw(George Jetson 35);

print OUTPUT join("|", @record), "\n";
```

## 3.3. Comma-separated variable (CSV) files

Comma separated variable files are another format commonly produced by spreadsheet and database programs. CSV files delimit their fields with commas, and wrap textual data in quotation marks, allowing the textual data to contain commas if required:

```
"Fred","Flintstone",40
"Wilma","Flintstone",36
"Barney","Rubble",38
"Betty","Rubble",34
"Homer","Simpson",45
"Marge","Simpson",39
"Bart","Simpson",11
"Lisa","Simpson",9
```

CSV files are harder to parse than ordinary delimited text files. The best way to parse them is to use the Text::ParseWords module:

```
#!/usr/bin/perl -w

use strict;
use Text::ParseWords;

open INPUT, "csv.txt" or die "Can't open input file: $!";

while (<INPUT>) {
my @fields = quotewords("," 0, $_);
}
```

The three arguments to the `quotewords()` routine are:

- The delimiter to use
- Whether to keep any backslashes that appear in the data (zero for no, one for yes)
- A list of lines to parse (in our case, one line at a time)

# 3.4. Problems with flat file databases

## 3.4.1. Locking

When using flat file databases without locking, problems can occur if two or more people open the files at the same time. This can cause data to be lost or corrupted.

If you are implementing a flat file database, you will need to handle file locking using Perl's `flock` function.

## 3.4.2. Complex data

If your data is more complex than a single table of scalar items, managing your flat file database can become extremely tedious and difficult.

## 3.4.3. Efficiency

Flat file databases are very inefficient for large quantities of data. Searching, sorting, and other simple activities can take a very long time and use a great deal of memory and other system resources.

# 3.5. Chapter summary

- The two main types of text database use either delimited text or comma separated variables to store data
- Delimited text can be read using Perl's `split` function and written using the `join` function

- Comma separated files are most easily read using the Text::ParseWords module
- There are several problems with flat file databases including locking, efficiency, and difficulties in handling more complex data

# Chapter 4. Relational databases

## 4.1. In this chapter...

The first section of this training session focuses on database theory, and covers relational database systems, and SQL - the language used to talk to them.

## 4.2. Tables and relationships

In a relational database, data is stored in tables. Each table contains data about a particular type of entity (either physical or conceptual).

For instance, our sample database is the inventory and sales system for Acme Widget Co. It has tables containing data for the following entities:

**Table 4-1. Acme Widget Co Tables**

| Table | Description |
|---|---|
| stock_item | Inventory items |
| customer | Customer account details |
| saleperson | Sales people working for Acme Widget Co. |
| sales | Sales events which occur |

Tables in a database contain fields and records. Each record describes one entity. Each field describes a single item of data for that entity. You can think of it like a spreadsheet, with the rows being the records and the columns being the fields, thus:

**Table 4-2. Sample table**

| ID number | Description | Price | Quantity in stock |
|---|---|---|---|
| 1 | widget | $9.95 | 12 |
| 2 | gadget | $3.27 | 20 |

Every table must have a *primary key*, which is a field which uniquely identifies the record. In the example above, the Stock ID number is the primary key.

The following figures show the tables used in our database, along with their field names and primary keys (in bold type).

**Table 4-3. the stock_item table**

| stock_item |
|---|
| *id* |
| description |
| price |
| quantity |

**Table 4-4. the customer table**

| customer |
|---|
| *id* |
| name |
| address |
| suburb |
| state |
| postcode |

**Table 4-5. the salesperson table**

| **salesperson** |
| --- |
| *id* |
| name |

**Table 4-6. the sales table**

| **sales** |
| --- |
| *id* |
| sale_date |
| salesperson_id |
| customer_id |
| stock_item_id |
| quantity |
| price |

# 4.3. Structured Query Language

SQL is a semi-English-like language used to manipulate relational databases. It is based on an ANSI standard, though very few SQL implementations actually adhere to the standard.

SQL statements are mostly case insensitive these days. While most books and references use upper-case, these notes use lower-case throughout for readability, and because the likelihood of needing to deal with older databases which only understand upper-case is becoming increasingly slim.

The syntax given in these coursenotes is cut down for simplicity; for full information, consult your database system's documentation. The MySQL documentation is available on our system in `/usr/doc/mysql-doc` and `/usr/doc/mysql-manual`, or by pointing your web browser at `http://training.netizen.com.au/`.

## 4.3.1. General syntax

SQL is case usually insensitive, apart from table and field names (which may or may not be case sensitive depending on what platform you're on -- on Unix they are usually case sensitive, on Windows they usually aren't).

String data can be delimited with either double or single quotes. Numerical data does not need to be delimited.

Wildcards may be used when searching for string data. A `%` (percent) sign is used to indicated multiple characters (much as an asterisk is used in DOS or Unix filename wildcards) while the underscore character (_) can be used to indicate a single character, similar to the `?` under Unix or DOS.

The following comparison operators may be used:

**Table 4-7. Comparison Operators**

| Operator | Meaning |
|---|---|
| = | Equality |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal to |
| <= | Less than or equal to |
| <> | Inequality |
| like | Wildcard matching |

In the following syntax examples, the term *`condition`* is used as shorthand for any expression which can be evaluated for truth, for instance `2 + 2 = 4` or `name like "A%"`.

Conditions may be combined by using `and` and `or`; use round brackets to indicate precedence. For instance, `name like "A%" or name like "B%"` will find all records where the "name" field starts with A or B.

#### 4.3.1.1. SELECT

An SQL `select` statement is used to select certain rows from a table or tables. A select query will return as many rows as match the criteria.

##### 4.3.1.1.1. Syntax

```
select field1 [, field2, field3] from table1 [, table2]
        where condition
        order by field [desc]
```

##### 4.3.1.1.2. Examples

```
select id, name from customer;
select id, name from customer order by name;
select id, name from customer order by name desc;
```

We can use a select statement to obtain data from multiple tables. This is referred to as a “join”.

```
select * from customer, sales where cus-
tomer.id = sales.customer_id
```

#### 4.3.1.2. INSERT

An `insert` query is used to add data to the database, a row at a time.

The columns names are optional to make typing queries easier. This is

fine for interactive use, however it is very bad practice to omit them in programs. *Always* specify column names in `insert` statements.

#### 4.3.1.2.1. Syntax

```
insert into tablename (col_name1, col_name2, col_name3) values (value1, value2, value3);
```

#### 4.3.1.2.2. Examples

```
insert into stock_item (id, description, price, quantity) values (0, 'doodad', 9.95, 12);
```

Note that since the `id` field is an `auto_increment` field in the Acme inventory database we've set up, we don't need to specify a value to go in there, and just use zero instead --- whatever we specify will be replaced with the auto-incremented value. Auto-increment fields of some kind are available in most database systems, and are very useful for creating unique ID numbers.

### 4.3.1.3. DELETE

A `delete` query can be used to delete rows which match a given criteria.

#### 4.3.1.3.1. Syntax

```
delete from tablename where condition
```

#### 4.3.1.3.2. Examples

```
delete from stock_item where quantity = 0;
```

## 4.3.1.4. UPDATE

The `update` query is used to change the values of certain fields in existing records.

### 4.3.1.4.1. Syntax

```
update tablename set field1 = expression, field2 = expression
        where condition
```

### 4.3.1.4.2. Examples

```
update stock_item set quantity = (quantity - 1) where id = 4;
```

## 4.3.1.5. CREATE

The `create` statement is used to create new tables in the database.

### 4.3.1.5.1. Syntax

```
create table tablename (
        column coltype options,
        column coltype options,
        ...
        primary key (colname)
)
```

Data types include (but are not limited to):

**Table 4-8. Some data types**

| | |
|---|---|
| INT | an integer number |
| FLOAT | a floating point number |

| | |
|---|---|
| CHAR(*n*) | character data of exactly *n characters* |
| VARCHAR(*n*) | character data of up to *n characters (field grows/shrinks to fit)* |
| BLOB | Binary Large OBject |
| DATE | A date in YYYY-MM-DD format |
| ENUM | enumerated string value (eg "Male" or "Female") |

Data types vary slightly between different database systems. The full range of MySQL data types is outlined in section 7.2 of the MySQL reference manual.

#### 4.3.1.5.2. Examples

```
create table contactlist (
        id int not null auto_increment,
        name varchar(30),
        phone varchar(30),
        primary key (id)
)
```

### 4.3.1.6. DROP

The `drop` statement is used to delete a table from the database.

#### 4.3.1.6.1. Syntax

```
drop table tablename
```

#### 4.3.1.6.2. Example

```
drop table contactlist
```

## 4.4. Chapter summary

- A database table contains fields and records of data about one entity
- SQL (Structured Query Language) can be used to manipulate and retrieve data in a database
- A `SELECT` query may be used to retrieve records which match certain criteria
- An `INSERT` query may be used to add new records to the database
- A `DELETE` query may be used to delete records from the database
- An `UPDATE` query may be used to modify records in the database
- A `CREATE` query may be used to create new tables in the database
- A `DROP` query may be used to remove tables from the database

# Chapter 5. MySQL

## 5.1. In this chapter...

In this section we examine the popular database MySQL, which is available for free for many platforms. MySQL is just one of many database systems which can be accessed via Perl's DBI module.

## 5.2. MySQL features

### 5.2.1. General features

- Fast
- Lightweight
- Command-line and GUI tools
- Supports a fairly large subset of SQL, including indexing, binary objects (BLOBs), etc
- Allows changes to structure of tables while running
- Wide userbase
- Support contracts available

### 5.2.2. Cross-platform compatibility

- Available for most Unix platforms

- Available for Windows NT/95/98 (there are license differences)
- Available for OS/2
- Programming libraries for C, Perl, Python, PHP, Java, Delphi, Tcl, Guile (a scheme interpreter), and probably more...
- Open-source ODBC

# 5.3. Comparisions with other popular DBMSs

## 5.3.1. PostgreSQL

MySQL and PostgreSQL are very similar in many ways. The main differences are that PostgreSQL is an object database system rather than a purely relational database system, it has transactions (but its performance suffers because of this) and that PostgreSQL is distributed under the GNU General Public License (GPL) rather than a license which imposes some restrictions or costs on use and redistribution.

More information: `http://www.postgresql.org/`

## 5.3.2. mSQL

mSQL is also similar to MySQL but has slightly less features and is not free for commercial use. On the positive side, it is very lightweight and can be very fast for simple SELECT queries.

More information: `http://www.hughes.com.au/`

## 5.3.3. Oracle, Sybase, etc

MySQL will not give you the performance or features of Oracle or other

enterprise-level database management systems. In particular, MySQL lacks transactions, triggers, and views. The price you pay for this is that Oracle costs a lot, and requires heavy hardware to run on. MySQL is better suited to small-to-medium database applications such as web-based database applications, and will do so happily on a common Pentium based system.

More information: `http://www.oracle.com/`

# 5.4. Getting MySQL

MySQL can be downloaded from `http://www.mysql.com/`or mirror sites worldwide. It is also available in packaged binary format for various operating system distributions, including RedHat and Debian linux.

Installation instructions come with the software, but in brief:

## 5.4.1. Redhat Linux

Download the appropriate RPM packages, and type `rpm -i` *`packagename`*`.rpm`

## 5.4.2. Debian Linux

Use `apt-get`, `dselect`, or `dpkg` to install the `.deb` packages. For instance, `apt-get install mysql`.

## 5.4.3. Compiling from source

Download the `tar.gz` file from `http://www.mysql.com/` and read the `README` file. Then type `./configure`, `make`, and `make install`.

### 5.4.4. Binaries for other platforms

Binaries are available for many platforms, including Windows and some commercial Unix platforms. Follow the installation instructions found in the `README` file.

## 5.5. Setting up MySQL databases

A tool called `mysqladmin` is distributed with MySQL. This tool allows the database administrator (DBA) to create, remove, or otherwise manage databases.

**Table 5-1. Mysqladmin commands:**

| | |
|---|---|
| create *`databasename`* | Create a new database |
| drop *`databasename`* | Delete a database and all its tables |
| flush-hosts | Flush all cached hosts |
| flush-logs | Flush all logs |
| flush-tables | Flush all tables |
| kill *`id,id,...`* | Kill mysql threads |
| password *`new-password`* | Change old password to new-password |
| processlist | Show list of active threads in server |
| reload | Reload grant tables |
| refresh | Flush all tables and close and open logfiles |
| shutdown | Take server down |
| status | Gives a short status message from the server |
| variables | Prints variables available |
| version | Get version info from server |

More help for this command is available by typing `mysqladmin --help` from the command line or by reading the MySQL reference manual.

## 5.5.1. Creating the Acme inventory database

To create a database called `inventory`, we would perform the following steps as the user who has permission to run `mysqladmin` (eg root):

```
% mysqladmin create inventory
% mysqladmin reload
```

## 5.5.2. Setting up permissions

To set up security permissions for the inventory database, we would need to create appropriate records in the `mysql` database (that's right, it's a database which has the same name as the database server). This is the central repository for access control information for all databases served by your MySQL server.

Typically, you will want to:

- create an entry in the `db` table for the database
- set the default permissions for the database
- create an entry in the `user` table for any users who should be allowed to access the database
- set default permissions for each user

All these are achieved by performing simple INSERT or UPDATE queries on the tables in question.

**Table 5-2. Available permissions include ...**

| | |
|---|---|
| Select | May perform SELECT queries |

| | |
|---|---|
| Insert | May perform INSERT queries |
| Update | May perform UPDATE queries |
| Delete | May perform DELETE queries |
| Create | May create new tables |
| Drop | May drop (delete) tables |
| Reload | May reload the database |
| Shutdown | May shut down the database |
| Process | Has access to processes on the OS |
| File | Has access to files on the OS's file system |

## 5.5.3. Creating tables

The SQL statements used to create tables are documented in the MySQL manual. `CREATE` statements are used to create each individual table by specifying the fields for each table, their data types and other options.

Below is an example --- these SQL statements create the Acme Widget Co. tables we will be working with throughout this session. The output you see is generated by the **mysqldump** program, and can be read back into a database via command line redirection, eg **mysql *`database`*** < ***`filename`***.

```
#
# Table structure for table 'customer'
#
CREATE TABLE customer (
  id int(11) DEFAULT '0' NOT NULL auto_increment,
  name varchar(80),
  address varchar(255),
  suburb varchar(50),
  state char(3),
  postcode char(4),
```

```
  PRIMARY KEY (id)
);

#
# Table structure for table 'sales'
#
CREATE TABLE sales (
  id int(11) DEFAULT '0' NOT NULL auto_increment,
  sale_date date,
  customer_id int(11),
  salesperson_id int(11),
  stock_item_id int(11),
  quantity int(11),
  price float(4,2),
  PRIMARY KEY (id)
);

#
# Table structure for table 'salesperson'
#
CREATE TABLE salesperson (
  id int(11) DEFAULT '0' NOT NULL auto_increment,
  name varchar(80),
  PRIMARY KEY (id)
);

#
# Table structure for table 'stock_item'
#
CREATE TABLE stock_item (
  id int(11) DEFAULT '0' NOT NULL auto_increment,
  description varchar(80),
  price float(4,2),
  quantity int(11),
  PRIMARY KEY (id)
);
```

# 5.6. The MySQL client

To talk to any database server, you will need to use a client of some kind. MySQL comes with a text-based client by default, but there are graphical clients available, as well as ODBC drivers to allow you to interact with a MySQL database from Windows applications such as Microsoft Access.

The command line client can be invoked from the command line with the `mysql` command. The `mysql` command takes a database name as a required argument, as well as other optional arguments such as `-p`, which causes the client to ask for a password for access to the database if access controls have been set up.

You can see all the options available on the command line by typing `mysql --help`.

You can set up access controls on a database by editing the data in the `mysql` database (i.e. type `mysql mysql` on the command line) or by using the `mysqlaccess` command. Type `mysqlaccess --help` for more information about this command.

```
% mysql -p databasename
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2 to server version: 3.21.33b

Type 'help' for help.

mysql>
```

The MySQL client allows you to type in commands on one or many lines. When you finish a statement, type `;` to end, same as for Perl.

To quit the client, type `quit` or `\q`.

For a full outline of commands available in the client, type `help` or `\h`. It will give you this message:

```
mysql> \h

MySQL commands:
help    (\h)    Display this text
?       (\h)    Synonym for 'help'
```

```
clear   (\c)    Clear command
connect (\r)    Reconnect to the server. Optional argu-
ments are db and host
edit    (\e)    Edit command with $EDITOR
exit    (\)     Exit mysql. Same as quit
go      (\g)    Send command to mysql server
print   (\p)    print current command
quit    (\q)    Quit mysql
rehash  (\#)    Rebuild completion hash
status  (\s)    Get status information from the server
use     (\u)    Use an-
other database. Takes database name as argument

Connection id: 1  (Can be used with mysqladmin kill)
```

# 5.7. Understanding the MySQL client prompts

The prompt that shows when you are using the MySQL client tells you a lot about what's going on.

The normal prompt looks like this:

```
mysql>
```

This means it is waiting for you to enter an SQL query.

If you are in the middle of entering an SQL query, it will be waiting for a semi-colon to terminate the query, and will look like this:

```
    ->
```

If you have opened a set of quotes but not closed them, you will see one of these prompts:

```
    '>
    ">
```

## 5.8. Exercises

1. Connect to a database which has the same name as your login (for instance, `train01`) by typing **mysql -p train01** (the `-p` flag causes it to ask you for your password, which in this case is the same as your login password). The database you are connecting to is your own personal copy of the Acme Widget Co. inventory and sales database mentioned in the previous section
2. Type `show tables` to show a list of tables in this database
3. Type `describe customer` to see a description of the fields in the table `customer`
4. Type `select * from customer` to perform a simple SQL query
5. Try selecting fields from other tables. Try both `select *` and `select` *`field1, field2`* type queries.
6. Use the `where` clause to limit which records you select
7. Use the `order by` clause to change the order in which records are returned
8. Insert a record into the customer table which contains your own name and address details
9. Update the price of widgets in the `stock_item` table to change their price to $19.95

When developing database applications, it is often useful to keep a client program such as this one open to test queries or check the state of your data. You can open multiple telnet sessions to our training system to do this if you wish.

## 5.9. Chapter summary

- MySQL is one of many database systems which can be used as the back-end to a

web site

- MySQL can be downloaded from http://www.mysql.com/ or mirror sites
- The MySQL command line client can be used to interact with MySQL databases
- The MySQL client allows the user to type in SQL queries and prints results to the screen.

# Chapter 6. The DBI and DBD modules

## 6.1. In this chapter...

In this section we look at the Perl module which can be used to interact with many database servers: DBI.

## 6.2. What is DBI?

Like the Perl modules discussed in last week's CGI programming course, the DBI and DBD modules are written by Perl people and distributed free via CPAN (the Comprehensive Perl Archive Network).

DBI stands for "Database Interface" while DBD stands for "Database Driver". You need both types of modules, working together, in order to access databases using Perl.

Modules are discussed in chapter 5 of the Camel, and documentation for the standard library modules is in chapter 7.

## 6.3. Supported database types

Databases supported by Perl's DBI module include:

- Oracle
- Sybase
- Informix
- MySQL
- Msql

- Ingres
- Postgres
- Xbase
- DB2
- ... and more

## 6.4. How does DBI work?

DBI is a generic interface which acts as a "funnel" between the programmer and multiple databases.

DBI protects you from needing to know the minutiae of connecting to different databases by providing a consistent interface for the programmer. The only thing you need to vary is the connection string, to indicate what sort of database you wish to connect to.

To use DBI, you need to install the DBI module from CPAN, as well as any DBD modules for the databases you use. For instance, to use MySQL you need to install the `DBD::Mysql` module.

To install DBI, download the DBI module from CPAN (http://www.perl.com/CPAN), unzip it using a command like **tar -xzvf DBI.tar.gz**, then follow the instructions in the **README** file distributed with the module.

## 6.5. DBI/DBD syntax

The syntax of the database modules is best found by using the **perldoc** command. **perldoc DBI** will give you general information applicable to all DBI scripts, while **perldoc DBD::*yourdatabase*** will give information specific to your own database. In our case, we use **perldoc DBD::mysql**.

DBI is an object oriented Perl module, like the `Text::Template` and `Mail::Mailer` modules covered in the CGI Programming in Perl training module. This means that when we connect to the database we will be creating an object which is called a "database handle" which refers to a specific session with the database. Thus we can have multiple sessions open at once by creating multiple database handles.

We can also create statement handle objects, which are Perl objects which refer to a previously prepared SQL statement. Once we have a statement handle, we can use it to execute the underlying SQL as often as we want.

### 6.5.1. Variable name conventions

The following variable name conventions are used in the DBD/DBI documentation:

**Table 6-1. DBI module variable naming conventions**

| Variable name | Meaning |
|---|---|
| `$dbh` | database handle object |
| `$sth` | statement handle object |
| `$rc` | Return code (boolean: true=ok, false=error) |
| `$rv` | Return value (usually an integer) |
| `@ary` | List of values returned from the database, typically a row of data |
| `$rows` | Number of rows processed (if available, else -1) |

## 6.6. Connecting to the database

```
use DBI;
```

```
my $driver = 'mysql';
my $database = 'database_name';         # name of your database here
my $username = undef;                   # your database username
my $password = undef;                   # your database password

# note that username and password should be as-
signed to if your database
# uses authentication (ie requires you to log in)

# we set up a connection string specific to this database
my $dsn = "DBI:$driver:database=$database";

# make the actual connection -
this returns a database handle we can use later
my $dbh = DBI->connect($dsn, $username, $password);

# when you're done (at the end of your script)
$dbh->disconnect();
```

## 6.7. Executing an SQL query

```
# set up an SQL statement
my $sql_statement = "select * from customer";
my $sth = $dbh->prepare($sql_statement)
|| die "Could not prepare: " . $dbh->errstr();

# execute it
$sth->execute() || die "Could not execute: " . $dbh->errstr();

# how many rows did we get?
my $num_rows = $sth->rows();
my $num_fields = $sth->{'NUM_OF_FIELDS'};

# close the sql query, if we don't want it any more.
```

```
$sth->finish();
```

## 6.8. Doing useful things with the data

```
# get an ar-
ray full of the next row of data that matches the query
# (the most common, and simplest, case)
while (my @ary = $sth->fetchrow_array()) {
        print "The first field is $ary[0]\n";
}

# get a hash reference instead
# (the more complicated, but more useful, version)
while (my $hashref= $sth->fetchrow_hashref()) {
        print "Name is $hashref->{'name'}\n";
}

# you can also get an arrayref
# (equally complicated and not quite as useful)
while (my $ary_ref = $sth->fetchrow_arrayref()) {
print "The first field is $ary_ref->[0]\n";
}
```

Of the above methods, `fetchrow_array()` is the only one that does not require an understanding of Perl references. References are not a beginner-level topic, but for those who are interested, they are documented in chapter 4 of the Camel. They are worth learning if only for the added benefit of being able to access fields by name when using the `fetchrow_hashref` method.

## 6.9. An easier way to execute non-SELECT queries

If you wish to execute a query such as INSERT, UPDATE, or DELETE, you may find it easier to use the `do()` method:

```
$dbh->do("delete from sales")
    || warn("Can't delete from sales table");
```

This method returns the number of rows affected, or undef if there is an error.

## 6.10. Quoting special characters in SQL

Sometimes you want to use a value in your SQL which may contain characters which have special behaviour in SQL, such as a percent sign or a quote mark. Luckily, there is a method which can automatically escape all special characters:

```
my $string = "20% off all stock";
my $clean_string = $dbh->quote($string);
```

## 6.11. Exercises

1. Use `exercises/scripts/easyconnect.pl` to connect to your Acme Widget Co. database. You will need to edit some of the lines at the top.
2. Use a `while` loop to output data a row at a time
3. Check all your statements for indications of failure, and output messages to the user using `warn()` if any of the steps fail.

### 6.11.1. Advanced exercises

1. If you wish, you can use a hash reference instead of an array
2. Change the SQL in `easyconnect.pl` to use a non-SELECT statement, and use the `do` method instead of the `prepare` and `execute` methods. Don't forget to check the return value!

## 6.12. Chapter summary

- The DBI module provides a consistent interface to a variety of database systems
- The DBI module can be downloaded from CPAN
- Documentation for the DBI module can be found by typing **perldoc DBI**

# Chapter 7. Acme Widget Co. Exercises

## 7.1. In this chapter...

In the second half of this training module, we will be tying together what we have learnt about SQL and DBI, and creating a simple application for Acme Widget Co. to assist them in inventory management, sales, and billing.

## 7.2. The Acme inventory application

In your `exercises/` directory you will find a subdirectory called `acme/` which contains the outline of the Acme inventory application which you will build upon for the rest of today.

## 7.3. Listing stock items

The shell of a stock-listing script is available in your `exercises/acme/` directory as `stocklist.pl`.

```
#!/usr/bin/perl -w
use strict;
use DBI;

my $driver = 'mysql';
my $database = 'trainXX';
my $username = 'trainXX';
my $password = 'your_password_here';

my $dsn = "DBI:$driver:database=$database";
my $dbh = DBI->connect($dsn, $username, $password)
```

```
        || die $DBI::errstr;

my $sql_statement = "select * from stock_item";
my $sth = $dbh->prepare($sql_statement);
$sth->execute() or die ("Can't execute SQL: " . $dbh->errstr());

while (my @ary = $sth->fetchrow_array()) {
        print <<"END";
ID:            $ary[0]
Description:   $ary[1]
Price:         $ary[2]
Quantity:      $ary[3]
END
}

$dbh->disconnect();
```

1. Fill in the variables indicated (`$database`, `$sql_statement`, etc)
2. Test your script from the command line
3. Sort the output in alphabetical order by Description

## 7.3.1. Advanced exercises:

1. If you are familiar with Perl references, convert the script to use `fetchrow_hashref()`
2. Ask the user to specify a field to sort by, either as a command line argument or on STDIN. If the sort order parameter is given, use it to change the sort order in your SQL statement and re-output the result, otherwise default to something sensible such as ID

# 7.4. Adding new stock items

1. Write a script which prompts the user for input, asking for values for description, quantity and price. Remember that the stock item's ID will be automatically filled in by the database, as it is an "auto increment" field.
2. Next, create an SQL query to add a record to the database. Output a message to the user indicating the success (or failure) of the operation. A sample script to get you started is available in `exercises/acme/addstock.pl`

## 7.4.1. Advanced exercises

1. Check that the price is a number (use regular expressions for these checks)
2. Check that it has two decimal places
3. Check that the number of items in stock is a number

# 7.5. Entering a sale into the system

1. The program `exercises/acme/sale.pl` provides an interface which can be used to input data pertinent to the occurence of a sale
2. Write a script which records the sale in the `sales` table
3. Your script will also have to update the `stock_item` table to reduce the number of items still in stock.
4. What happens if you try to buy/sell more items than are available? Put in a check to stop this from happening.

# 7.6. Creating sales reports

1. Copy the code from the previous example’s script to create a script that asks the user for a salesperson’s ID number and a start and end date.
2. Use the script to output a sales report for the chosen salesperson for the period between the two dates.

## 7.6.1. Advanced exercises

1. Create an extra option for "all" sales people, which shows all the sales people in descending order of sales made. You may need to use an SQL `group by` clause to achieve this.

# 7.7. Searching for stock items

1. Create a script which asks a user for a string to search for in a stock item’s description (eg "dynamite").
2. Allow the user to choose either "Full name", "Beginning of name" or "Part of name" as a search type.
3. Create different SQL queries using `LIKE` to search the data depending on their choices

### 7.7.1. Advanced exercises

1. Change the script so that people can use DOS/Unix style wildcards (`*` and `?`) then use their wildcard expression in your SQL query - convert the wildcards to SQL-style wildcards by using regular expressions

# Chapter 8. References (Optional topic)

## 8.1. In this chapter...

This section is included as an optional topic. It is intended for those who have experience in C or other languages which use pointers and references.

As mentioned earlier, references are covered in chapter 4 of the Camel. They are also covered at length in the first chapter of the O'Reilly book "Advanced Perl Programming" by Sriram Srinivasan (the "Panther" book). Lastly, **perldoc perlref** contains online documentation related to Perl references.

Uses for Perl references:

- creating complex data structures, for example multi-dimensional arrays
- passing multiple arrays and hashes to subroutines and functions without them getting smushed together
- creating anonymous data structures

## 8.2. Creating and deferencing

To create a reference to a scalar, array or hash, we prefix its name with a backslash:

```
my $scalar = "This is a scalar";
my @array  = qw(a b c);
my %hash   = ('sky' => 'blue', 'ap-
ple' => 'red', 'grass' => 'green');

my $scalar_ref = \$scalar;
my $array_ref  = \@array;
my $hash_ref   = \%hash;
```

Note that all references are scalars, because they contain a single item of information - the memory address of the actual data.

Dereferencing (getting at the value that a reference points to) is achieved by prepending the appropriate variable-type punctuation to the name of the reference. For instance, if we have a hash reference `$hash_reference` we can dereference it by looking for `%$hash_reference`.

```
my $new_scalar = $$scalar_ref;
my @new_array  = @$array_ref;
my %new_hash   = %$hash_ref;
```

In other words, wherever you would normally put a variable name (like `new_scalar`) you can put a reference variable (like `$scalar_ref`).

Here's how you access array elements or slices, and hash elements:

```
print $$array_ref[0];           # prints the first ele-
ment of the array
                                # referenced by $array_ref
print $$array_ref[0..2];        # prints an array slice
print $$hash_ref{'sky'};        # prints a hash element's value
```

The other way to access the value that a reference points to is using the "arrow" notation. This notation is usually considered to be better Perl style than the one shown above, which can have precedence problems and is less visually clean.

```
print $array_ref->[0];
print $hash_ref->{'sky'};
```

The Panther book describes a good way to visualise this method. Ask your instructor to demonstrate it or to loan you a copy of the book if you need a better understanding of the above syntax.

## 8.3. Complex data structures

We can use references to create complex data structures, such as this hash in which the values are arrays rather than scalars. Actually, they are scalars, since the array references are scalars, but they point to arrays.

```
my @fruits = qw(apple orange pear banana);
my @rodents = qw(mouse rat hamster gerbil rabbit);
my @books = qw(camel llama panther);

my %categories = (
        'fruits'        =>              \@fruits,
        'rodents'       =>              \@rodents,
        'books'         =>              \@books,
);

# to print out "gerbil"...
print $categories->{'rodents'}->[3];
```

## 8.4. Passing multiple arrays/hashes as arguments

If we were to attempt to pass two arrays together to a function or subroutine, they would be flattened out to form one large array:

```
mylist(@fruit, @rodents);

# print out all the fruits then all the rodents
sub mylist {
        my @list = @_;
        foreach (@list) {
                print "$_\n";
        }
}
```

If we want them kept separate, pass references:

```
myreflist(@fruit, @rodents);

sub myreflist {
        my ($firstref, $secondref) = @_;
        print "First list:\n";
        foreach (@$firstref) {
                print "$_\n";
        }
        print "Second list:\n";
        foreach (@$secondref) {
                print "$_\n";
        }
}
```

# 8.5. Anonymous data structures

Lastly, references can be used to create anonymous data structures which are destroyed once you’re done with them. An anonymous array is created by using square brackets instead of round ones. An anonymous hash uses curly brackets instead of round ones.

```
# the old two-step way:
my @array = qw(a b c d);
my $array_ref = \@array;

# if we get rid of $array_ref, @array will still hang round us-
ing up
# memory.  Here’s how we do it without the intermediate step:

my $array_ref = [’a’, ’b’, ’c’, ’d’];

# look, we can still use qw() too...

my $array_ref = [qw(a b c d)];
```

```
# more useful yet:

my %transport = (
        'cars'          =>      [qw(toyota ford holden porsche)],
        'planes'        =>      [qw(boeing harrier)],
        'boats'         =>      [qw(clipper skiff dinghy)],
);
```

## 8.6. Chapter summary

- References may be used to create complex data structures, pass multiple arrays and hashes to subroutines, and to create anonymous data structures
- References are created by prefixing the name of a variable with a backslash
- References are dereferenced by using the name of a reference (including the dollar sign) where we would usually use the alphanumeric name of a variable, or by using the arrow notation.
- References can be included in Perl data structures anywhere you might ordinarily find scalars.
- References to anonymous arrays may be created by initialising an array using square brackets instead of round ones.
- References to anonymous hashes may be created by initialising an hash using curly brackets instead of round ones.

# Chapter 9. Conclusion

## 9.1. What you've learnt

Now you've completed Netizen's Database Programming with Perl module, you should be confident in your knowledge of the following fields:

- Database terminology, including tables and relationships, fields and records, etc
- Flat file database manipulation including delimited and CSV text files
- Basic SQL queries, including `SELECT`, `INSERT`, `DELETE`, and `UPDATE` queries
- Features of MySQL, where to get MySQL from, and how to set up MySQL databases
- Using the MySQL command line client to perform SQL queries
- Using Perl's DBI module to interact with databases
- Applying Perl skills from previous training modules to create database applications

## 9.2. Where to now?

To further extend your knowledge of Perl, you may like to:

- Borrow or purchase the books listed in our "Further Reading" section (below)
- Follow some of the URLs given throughout these course notes, especially the ones marked "Readme"
- Install Perl and MySQL (or other database servers) on your home or work computer
- Practice using Perl to interact with databases
- Join a Perl user group such as Perl Mongers (http://www.pm.org/)

Information about other Netizen courses can be found on Netizen's website (http://netizen.com.au/services/training/). A diagram of Netizen's courses and the careers they can lead to is included with these training materials.

# 9.3. Further reading

## 9.3.1. Books

- Alligator Descartes & Tim Bunce, "Programming the Perl DBI", O'Reilly and Associates, 2000
- Randy Jay Yarger, George Reese & Tim King, "mSQL and MySQL", O'Reilly and Associates, 1999

## 9.3.2. Online

- The Perl homepage (http://www.perl.com/)
- The Perl Journal (http://www.tpj.com/)
- Perlmonth (http://www.perlmonth.com/) (online journal)
- Perl Mongers Perl user groups (http://www.pm.org/)
- comp.lang.perl.announce newsgroup
- comp.lang.perl.moderated newsgroup
- comp.lang.perl.misc newsgroup

# Appendix A. Unix cheat sheet

A brief run-down for those whose Unix skills are rusty:

**Table A-1. Simple Unix commands**

| Action | Command |
|---|---|
| Change to home directory | **cd** |
| Change to *directory* | **cd *directory*** |
| Change to directory above current directory | **cd ..** |
| Show current directory | **pwd** |
| Directory listing | **ls** |
| Wide directory listing, showing hidden files | **ls -al** |
| Showing file permissions | **ls -al** |
| Making a file executable | **chmod +x *filename*** |
| Printing a long file a screenful at a time | **more *filename* or less *filename*** |
| Getting help for *command* | **man *command*** |

# Appendix B. Editor cheat sheet

This summary is laid out as follows:

**Table B-1. Layout of editor cheat sheets**

| | |
|---|---|
| Running | Recommended command line for starting it. |
| Using | Really basic howto. This is not even an attempt at a detailed howto. |
| Exiting | How to quit. |
| Gotchas | Oddities to watch for. |

## B.1. vi

### B.1.1. Running

```
% vi filename
```

### B.1.2. Using

- `i` to enter insert mode, then type text, press **ESC** to leave insert mode.
- `x` to delete character below cursor.
- `dd` to delete the current line
- Cursor keys should move the cursor while *not* in insert mode.
- If not, try `hjkl`, `h` = left, `l` = right, `j` = down, `k` = up.

- `/`, then a string, then **ENTER** to search for text.
- `:w` then **ENTER** to save.

## B.1.3. Exiting

- Press **ESC** if necessary to leave insert mode.
- `:q` then **ENTER** to exit.
- `:q!` **ENTER** to exit without saving.
- `:wq` to exit with save.

## B.1.4. Gotchas

**vi** has an insert mode and a command mode. Text entry only works in insert mode, and cursor motion only works in command mode. If you get confused about what mode you are in, pressing **ESC** twice is guaranteed to get you back to command mode (from where you press i to insert text, etc).

## B.1.5. Help

`:help` **ENTER** might work. If not, then see the manpage.

# B.2. pico

## B.2.1. Running

```
% pico -w filename
```

## B.2.2. Using

- Cursor keys should work to move the cursor.
- Type to insert text under the cursor.
- The menu bar has `^X` commands listed. This means hold down **CTRL** and press the letter involved, eg **CTRL-W** to search for text.
- **CTRL-O**to save.

## B.2.3. Exiting

Follow the menu bar, if you are in the midst of a command. Use **CTRL-X** from the main menu.

## B.2.4. Gotchas

Line wraps are automatically inserted unless the -w flag is given on the command line. This often causes problems when strings are wrapped in the middle of code and similar. \\ \hline

### B.2.5. Help

**CTRL-G** from the main menu, or just read the menu bar.

## B.3. joe

### B.3.1. Running

```
% joe filename
```

### B.3.2. Using

- Cursor keys to move the cursor.
- Type to insert text under the cursor.
- **CTRL-K** then **S** to save.

### B.3.3. Exiting

- **CTRL-C** to exit without save.
- **CTRL-K** then **X** to save and exit.

### B.3.4. Gotchas

Nothing in particular.

### B.3.5. Help

**CTRL-K** then **H**.

## B.4. jed

### B.4.1. Running

```
% jed
```

### B.4.2. Using

- Defaults to the emacs emulation mode.
- Cursor keys to move the cursor.
- Type to insert text under the cursor.
- **CTRL-X** then **S** to save.

### B.4.3. Exiting

**CTRL-X** then **CTRL-C** to exit.

### B.4.4. Gotchas

Nothing in particular.

### B.4.5. Help

- Read the menu bar at the top.
- Press **ESC** then **?** then **H** from the main menu.

# Appendix C. ASCII Pronunciation Guide

**Table C-1. ASCII Pronunciation Guide**

| Character | Pronunciation |
|---|---|
| ! | bang, exlamation |
| * | star, asterisk |
| $ | dollar |
| @ | at |
| % | percent |
| & | ampersand |
| " | double-quote |
| ' | single-quote, tick |
| ( ) | open/close bracket, parentheses |
| < | less than |
| > | greater than |
| - | dash, hyphen |
| . | dot |
| , | comma |
| / | slash, forward-slash |
| \ | backslash, slosh |
| : | colon |
| ; | semi-colon |
| = | equals |
| ? | question-mark |
| ^ | caret (pron. carrot) |
| _ | underscore |

| Character | Pronunciation |
|---|---|
| [ ] | open/close square bracket |
| { } | open/close curly brackets, open/close brace |
| \| | pipe, or vertical bar |
| ~ | tilde (pron. “til-duh”, wiggle, squiggle) |
| ‘ | backtick |